1. (20%, a) (Maze Traversal) The following grid is a double-subscripted array representation of a maze.



The # symbols represent the walls of the maze, and the periods (.) represent squares in the possible paths through the maze.

There’s a simple algorithm for walking through a maze that guarantees finding the exit (assuming there’s an exit). If there’s not an exit, you’ll arrive at the starting location again. Place your right hand on the wall to your right and begin walking forward. Never remove your hand from the wall. If the maze turns to the right, you follow the wall to the right. As long as you do not remove your hand from the wall, eventually you’ll arrive at the exit of the maze. There may be a shorter path than the one you have taken, but you’re guaranteed to get out of the maze.

Write recursive function mazeTraverse to walk through the maze. The function should receive as arguments a 12-by-12 character array representing the maze and the starting location of the maze. As mazeTraverse attempts to locate the exit from the maze, it should place the character X in each square in the path. The function should display the maze after each move so the user can watch as the maze is solved.

(12.5%, b) (Generating Mazes Randomly) Write a function mazeGenerator that takes as an argument a double-subscripted 12-by-12 character array and randomly produces a maze. The function should also provide the starting and ending locations of the maze.

(12.5%, c) Write a test program to set up a 12-by-12 character array and try your function mazeGenerator and mazeTraverse. Show the maze and the path that you find.

1. ***(15%, Card Shuffling and Dealing Modification)*** In the card shuffling and dealing program of Fig. 7.24, we intentionally used an inefficient shuffling algorithm that introduced the possibility of indefinite postponement. In this problem, you’ll create a high-performance shuffling algorithm that avoids indefinite postponement. Modify the program of Fig. 7.24 as follows. Begin by initializing the deck array as shown in Fig. 7.29. Modify the shuffle function to loop row-by-row and column-by-column through the array, touching every element once. Each element should be swapped with a randomly selected element of the array.

Print the resulting array to determine if the deck is satisfactorily shuffled (as in Fig. 7.30, for example). You may want your program to call the **shuffle** function several times to ensure a satisfactory shuffle.

Note that although the approach in this problem improves the shuffling algorithm, the dealing algorithm still requires searching the deck array for card 1, then card 2, then card 3, and so on. Worse yet, even after the dealing algorithm locates and deals the card, the algorithm continues searching through the remainder of the deck. Modify the program of Fig. 7.24 so that once a card is dealt, no further attempts are made to match that card number, and the program immediately proceeds with dealing the next card.



1. **(20%, Card Shuffling and Dealing)** Modify the card shuffling and dealing program so that the card-dealing function deals a five-card poker hand. Then write the following additional functions:
2. Determine whether the hand contains a pair
3. Determine whether the hand contains two pairs
4. Determine whether the hand contains three of a kind (e.g., three jacks)
5. Determine whether the hand contains four of a kind (e.g., four aces)
6. Determine whether the hand contains a flush (i.e., all five cards of the same suit)
7. Determine whether the hand contains a full house
8. Determine whether the hand contains a straight (i.e., five cards of consecutive face values)

Show the dealt poker cards and check them by the functions. Print out the check results if the poker cards satisfy any patterns of the above defined poker hands.



1. **(20%, Project: Card Shuffling and Dealing)** Use the functions developed in problem 2 to write a program that deals two five-card poker hands, evaluates each, and determines which is the better hand.